Prácticas Compiladores Generación Código

Grupos de Kostadin Koruchev.

Como SI y como NO hay que generar el código

En general: SI después de leer un libro sobre el tema (incluido yacc) y NO antes de hacerlo. Los aspectos en este material son críticos. Si no entendéis algo – pedir tutoría.

1 Expresiones aritméticas:

Se suponen las siguientes declaraciones:

```
int a,aa;
ref int b,bb;
ref ref int c,cc;
bool x,xx;
ref bool y,yy;
ref ref bool z,zz;
```

1.1 Ámbito de nombres

Hay que hacer todos los identificadores distintos de las palabras claves del ensamblador.

— NO:

```
    ...
a   dd   0
    ...
; a := 5;
    push 5
    pop  eax
    mov  [a],eax ; NO! si a fuese and seria un error.
```

******* SI:**

```
    ...
_a  dd  0
    ...
; a := 5;
    push 5
    pop  eax
    mov  [_a],eax ; Cambio del ambito de nombres
```

1.2 Guardar el contexto del programa

El contexto del programa consiste de
`ebx,esi,edi,ebp,esp,eds` y si hay que cambiarlo hay que recuperarlo al salir de la función.
—— NO:

```
; a := 5;
    push 5
    pop  ebx ; Utilización de contexto
    mov  [_a],ebx
```

******* SI:**

```
; a := 5;
    push dword 5
    pop  eax
    mov [_a],eax
```

1.3 Precedencia de operadores

Ver el análisis de errores de la práctica anterior. La precedencia es
$'+' = '-' < '*' < ' -'$ *unario*.

1.4 Signed/unsigned

El juego de operadores signed (como regexp) es:
$imul, jn?[gl]e?, jn?e$

— NO:

```
; (-1 < 2)
    push dword -1
    push dword 2
    pop  eax
    pop  edx
    cmp  eax,edx
    jb   si222   ; Mezcla aritméticas
    mov  eax,0
    jmp  cp222
si222:
    mov eax,1
cp222:
```

Posible:

```
; (-1 < 2)
    push dword -1
    push dword 2
    pop  eax
    pop  edx
    cmp  eax,edx
    jl   si222   ; Aritmética con signo
    mov  eax,0
    jmp  cp222
si222:
    mov eax,1
cp222:
    push eax
```

***** **SI:**

```
; (-1 < 2)
    push  dword -1
    push  dword 2
    pop   eax
    pop   edx
    cmp   eax,edx
    setl  al        ; 386 no necesita saltos
    movxb eax,al
    push  eax
```

1.5 Dereferenciar

La asignación de id a id hay que tratarla como excepción, es decir añadirla como regla separada. Al tener dos posibilidades para la parte derecha, vamos a producir conflicto reduce-reduce. Hay que ordenar las sentencias de exp, asignaciones para que el conflicto se resuelve como lo necesitamos. Entonces R-value se dereferencia solo en $exp : id$ y L-value se dereferencia solo en $asign : id := id$ y de modo trivial en $input\ id$ y $asign : id := exp$. Se puede dereferenciar con cualquier registro de 386.

Un R-value:

```
; **c
    mov eax,[_c]
    mov eax,[eax]
    ; ... tantas veces eax <- [eax] cuantas *
    mov eax,[eax]
```

Un L-value trivial – lo mismo como R-value pero con un `mov eax,[eax]` menos.

El no trivial – según la explicación del semántica.

2 Una acción en un sitio:

Hay que evitar que una misma acción, salvo los más triviales aparezcan en dos sitios distintos.

NO:

```
exp: exp '+' exp { if (...) fprintf ( ... pop ... add ... }
   | exp '-' exp { if (...) fprintf ( ... pop ... sub ... }
```

******* SI:**

```
exp: exp '+' exp
{ $$.tipo=op_binaria($1.tipo,$3.tipo,''add'',''or''); }
   | exp '-' exp
{ $$.tipo=op_binaria($1.tipo,$3.tipo,''sub'',''xor''); }
```

3 L atributos:

No es recomendable la utilización de variables globales porque las gramáticas normalmente son recursivas. Para el analizador sintáctico son estrictamente necesarias 2 variables globales: el hash y el enumerador de las etiquetas. Si la gramática el localmente LL1 o L-atributiva (como es de ASPLE), hay que utilizar la pila con $PREV()$:

NO RECOMENDABLE:

```
/* variables globales */
mode : _int     { glb_tipo=ASP_INT;  glb_refs=0;}
     | _bool    { glb_tipo=ASP_BOOL; glb_refs=0;}
     | ref mode { glb_refs++; }
decl : mode idlist ';'
/* dos acciones iguales */
idlist : id            { wr_hash($1.texto,glb_tipo,glb_refs);}
       | id ',' idlist { wr_hash($1.texto,glb_tipo,glb_refs);}
```

UN POCO MEJOR:

```
/* variables globales */
mode : _int     { glb_tipo=ASP_INT;  glb_refs=0;}
     | _bool    { glb_tipo=ASP_BOOL; glb_refs=0;}
     | ref mode { glb_refs++; }
decl : mode idlist ';'
idlist : idunknown
       | idunknown ',' idlist
idunknown:  id         { wr_hash($1.texto,glb_tipo,glb_refs);}
```

***** **SI:**

```
mode : _int     { $$.tipo=ASP_INT;  $$.refs=0;}
     | _bool    { $$.tipo=ASP_BOOL; $$.refs=0;}
     | ref mode { $$=$2; $$.refs++; }
decl   : mode idlist ';'
idlist : idunknown
       | idunknown comma idlist
comma  : ','     { $$=$-1;}
idunknown : id   { wr_hash($1.texto,$0.tipo,$0.refs);}
```

Caso parecido – case.

4 Rango de saltos

NO:

```
; while (a=5) ... end
W_555:
    push dword 5
    push [_a]
    pop eax
    pop edx
    cmp eax,edx
    jne WE_555 ; este jump tiene rango limitado.
    ...
    jmp W_555
WE_555:
```

***** **SI:**

```
; while (a=5) ... end
W_555:
    push dword 5
    push [_a]
    pop eax
    pop edx
    cmp eax,edx
    je G_555 ; este jump tiene rango limitado.
    jmp WE_555
G_555:     ; y destino cercano
    ...
    jmp W_555
WE_555:
```

5 Desbalance de la pila

NO:

```
; output 5;
    push 5
    call output
    ...
```

******* SI:**

```
; output 5;
    push dword 5
    call output
    pop  eax ; o add esp,4
    ...
```

6 Tecnología

6.1 Tamaño del programa

El programa yacc (generación de código) es de unas 250-300 lineas. Si el tamaño del programa es significativamente mayor que éste, hay error de concepto o de estructura.

6.2 Plan de desarrollo

Hay que tener siempre un programa funcionando de todo y un plan claro de desarrollo. Un posible plan:

1. program – prolog y epilog en asm.
2. int constant, output, libreria IO (en C).
3. $+, -, *, -unario$ (int). Estructurado.
4. Polimorfismo: bool constant, $+, -, *$ general.
5. $<, >=, =$.
6. input. Comprobación de expresiones aritméticas.
7. Asignación(naive), if, while: comprobación con unos ejemplo de WEB.
8. Replica asignación, resolución de conflicto. asignación según el enunciado.
9. repeat until, procedure etc.
10. case. Tratar L-atributo.

11. Cambiar el generador de código de practica 3 para que genere ejemplos con semántica valido. Se puede hacer separando la gramática de exp a exp_{int} y exp_{bool}. Comprobar con programas aleatorios (solo si compila y linka).

12. Escribir programas de prueba pensando sobre los errores posibles. (Podéis intercambiar estos programas).

6.3 Comprobación con los ejemplos

Como principio de ingeniería general, al encontrar un error y corregirlo hay que descartar el ejemplo del conjunto de prueba y sustituirlo con uno parecido de tamaño y complexidad.

Si no se descarta el ejemplo ya utilizado, no aumenta la vericidad del programa. Ver un libro de texto de control de calidad.